4，栈和队列-2

- 栈和队列的概念
- 数据的生成，缓存，使用和顺序
- 栈的实现和问题
- 栈应用实例
- 栈与递归，递归和非递归
- 队列的实现和问题
- 队列应用实例
- 搜索问题
- 相关问题

栈与递归

- 如果一个定义或者结构（如 Python 函数，数据结构）中的某个或某几个部分具有与整体同样的结构，则称其为递归定义或递归结构
- 递归定义中的递归部分必须比整体简单，这样最后才能有终结点（称为递归定义的出口）；递归结构中也必须存在由非递归的基本结构构成的部分。否则就是无限递归，不是良好定义
- 例：
  - 递归定义的 Python 函数（所完成工作的一部分通过调用自身完成）
  - 结点链构成的单链表（非空时，去掉一个结点后还是同样结构）
- 例：简单表达式
  - 常数、变量是表达式；
  - 若 e_1, e_2 是表达式，op 为运算符，则

    e_1 op e_2， op e_1， (e_1) 也是表达式

栈与递归

- 例：阶乘函数　n!

$$fact(n) = \begin{cases} 1 & n=0 \\ n * fact(n-1) & n>0 \end{cases}$$

相应的 Python 函数定义（一种定义）：

```
def fact(n) :
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

- 递归算法（和递归定义的函数）很有用，主要适用于要解决的问题、要计算的函数、或者要处理的数据具有递归性质的情况
- 问题：在递归函数的执行中将会递归调用自己，而且还可能继续这样递归调用。这种过程在计算机上如何实现？

栈与递归

- 考虑递归定义的函数 fact

```
def fact(n) :
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

- 在递归调用中保存的数据，如 6, 5, ...，后保存的将先使用
- 后进先出的使用方式和数据项数无明确限制，说明应该用一个栈支持递归函数的实现

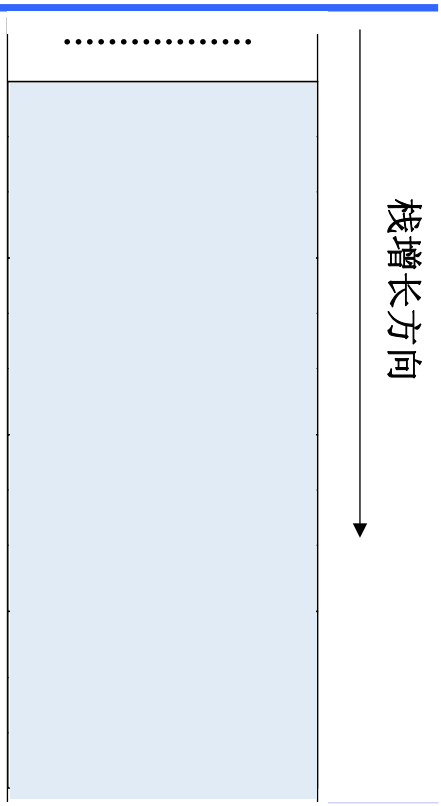
- 不难看到下面一些显然的情况（以 fact(6) 的计算为例）
  - 为得到 fact(6) 的结果，必须先算出 fact(5)
  - 在计算 fact(6) 时函数的参数 n 取值 6，而在递归调用计算 fact(5) 时，函数的参数 n 取值 5，如此下去
  - 递归调用计算出 fact(5) 的值之后还需乘以 6 以得到 fatc(6) 的值，这说明在递归调用 fact(5) 时 n 具有值 6 的情况需要记录（保存）
  - 需要这样记录的数据量与递归的次数成线性关系，无常量限制，因此不能用定义几个整型变量的方式保存

栈与递归

- 看阶乘函数导致的递归计算

```
def fact(n) :
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

- 假定现在执行 **fact(3)**，与此有关的实际数据共有三项
  - 计算结果
  - **n** 的值
  - 计算后回到的位置
- 递归调用 **fact(2)**，**fact(1)** 和 **fact(0)** 的情况都一样



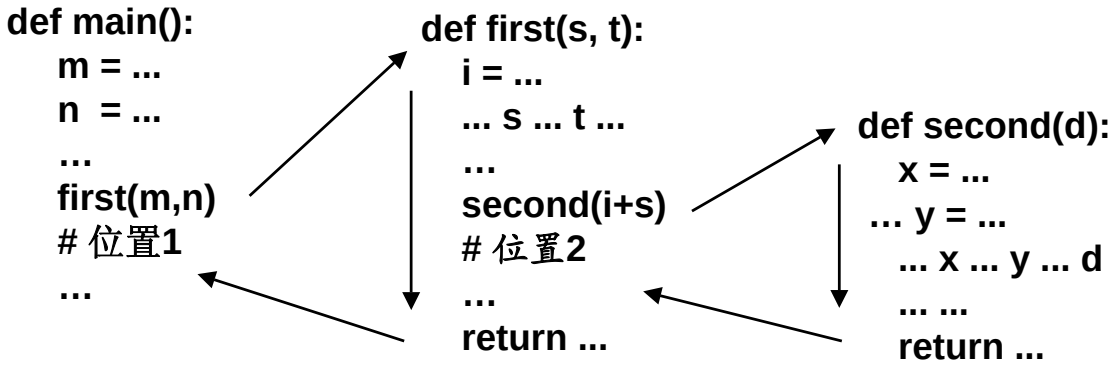
栈与递归/函数调用

- 支持递归的实现需要一个栈（运行栈），实现递归函数时
  - 每个具体递归调用都有一些局部信息需要保存，语言的实现在运行栈上为函数的这次调用建立一个帧，其中保存有关信息
  - 函数执行总以栈顶帧作为当前帧，所有局部变量都在这里有体现
  - 进入下次递归调用时，将为它建立一个新帧
  - 从递归调用返回时，上层取得函数调用的结果，并弹出已经结束的调用对应的帧，然后回到上一层执行时的状态

  所有递归程序的执行都是这种模式
- 实际上，一般的函数调用和退出的方式也与此类似
  - 目前各种语言都按这种模式实现了一套支持函数调用和返回的机制，其中最重要的数据结构就是一个运行栈
  - 其中保存所有已经开始（被调用）执行但还没有结束的函数的局部信息（局部变量的值约束等）

栈与函数调用

- 程序里函数嵌套调用是按“后调用先返回”的规则进行
  - 这种规则符合栈的使用模式
  - 用栈可以自然地支持函数调用的实现



- 支持函数调用的进行，语言实现需要做一些内部动作

  在进入新函数前保存一些信息，退出函数时恢复调用前状态并继续

栈与函数调用

- 这两部分动作分别称为函数调用的前序和后序动作
- 调用的前序动作：
  - 为被调用函数的局部变量分配存储区（函数帧/活动记录/数据区）
  - 将所有实参和返回地址存入函数帧（实参形参的结合/传值）
  - 将控制转到被调用函数入口
- 调用的后序动作（返回）：
  - 将被调用函数的计算结果存入指定位置；
  - 释放被调用函数的存储区；
  - 按以前保存的返回地址将控制转回调用函数
- 递归定义的函数每次递归函数调用，都将自动执行这些动作
  - 要想把递归定义的函数变换成非递归的，就需要自己做这些事情，用一个栈保存使用的中间信息

栈与函数调用

- 考虑阶乘函数的非递归形式，用自己定义的栈模拟系统的运行栈
- 函数定义：

```
def norec_fact (n): # 自己管理栈，模拟函数调用过程
    res = 1
    st = SStack();
    while n > 0:
        st.push(n)
        n -= 1
    while not st.is_empty():
        res *= st.pop()
    return res
```

- 这里并没有严格地按规矩翻译，只保存了必要的信息。

  例如这里的计算结果没有进栈

递归过程和非递归过程

- 前面先给求阶乘的递归算法，而后给出了一个使用栈保存搜索的中间信息的非递归算法
- 可以证明：任何递归定义的函数（程序），都可以通过引入一个栈保存中间结果，翻译为一个非递归的过程。与此对应，任何一个包含循环的程序都可翻译为一个不包含循环的递归程序

  这两个翻译过程都可计算，可以写出完成这两种翻译的程序，把任何递归定义的函数翻译到完成同样工作的非递归的函数，或者把任何包含循环的程序翻译为不包含循环的递归程序
- 阶乘的递归算法里只有一个递归调用，很容易翻译成非递归算法

  前面的翻译并没有采用通用的方法，而是根据自己对函数执行过程的分析，尽可能地做了些简化。这样写出的程序比较简单
- 有关自动翻译的算法，这里不再介绍

  有兴趣的同学可以参考张乃孝老师的《算法与数据结构》，高教出版社 **2002** 年版，**4.3.1** 节，或北大信科学院的数据结构幻灯片

栈的应用：简单背包问题

- 问题描述：背包里可放入重量为 W 的物品，现有 n 件物品的集合 S，其中物品的重量分别为 $w_1, w_2, \ldots, w_n$。问能否从中选出若干件物品的重量之和正好是 W。若存在则称此背包问题有解，否则无解
  - 可以要求当存在解时给出一个解
  - 许多实际的货物安排，装车，剪裁材料等，都与这一问题类似
- 问题的表示：设 W ≥ 0，n ≥ 0。用 knap(W, n) 表示 n 件物品相对于 W 的背包问题，如果它有解
  - 如果不选 w_n，则 knap(W, n - 1) 的解就是 knap(W, n) 的解
  - 如果选 w_n，则 knap(W - w_n, n - 1) 的解 就是 knap(W, n) 的解
- 问题有递归性质，n 件物品的背包问题可归结到 n - 1 件物品的问题
  - 可能是对另一个总容量 W
  - 通过不断归结，最后可以归结到最简单的情况

简单背包问题

- 背包问题的递归定义：

$$
\text{knap}(W, n) = \begin{cases} \text{True} & W = 0 \\ \text{False} & W < 0 \\ \text{False} & W > 0 \text{ , } n < 1 \\ \text{knap}(W, n-1) \text{ 或 } \text{knap}(W-w_n, n-1) & \text{当 } s > 0,\ n \geq 1 \end{cases}
$$

- 这里的 True 表示有解；False 表示无解
  - 前三种情况可以直接知道有没有解
  - 后两种情况都是把原问题归结到规模较小的问题。这样归结下去，最终会达到前三种情况
- 注意：每件物品有且仅有一件，用去了就没有了

简单背包问题

- 完全根据上面的递归定义，写出的递归函数定义如下：

```
def knap_rec( weight, wlist, n ):
    if weight == 0:
        return True
    elif weight < 0 or (weight > 0 and n < 1):
        return False
    elif knap_rec(weight - wlist[n-1], wlist, n-1):
        print("Item " + str(n) + ":", wlist[n-1])
        return True
    elif knap_rec(weight, wlist, n-1):
        return True
    else: return False
```

- 函数还产生了输出，列出所选的各物品的顺序号和重量

简单背包问题

- 对背包问题，有关算法里出现了两个递归调用

```
def knap_rec( weight, wlist, n ):
    if ... ...
    elif knap_rec(weight - wlist[n-1], wlist, n-1):
        print("Item " + str(n) + ":", wlist[n-1]); return True
    elif knap_rec(weight, wlist, n-1): return True
    else: ...
```

- 按规范方式翻译得到的非递归函数定义比较长，这里不讨论了

  有兴趣的同学可以查阅张乃孝老师主编的《算法与数据结构》，高教出版社，2002 版，4.3.1 节，或者信息学院的数据结构幻灯片

  可以自己想想如何写出一个（简单些的）非递归算法（作为自由练习，可以在教学网的课程讨论组交流）

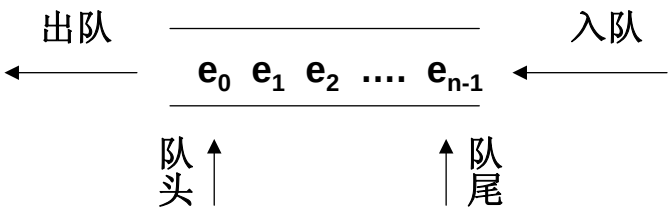
- 在现在的计算机上，函数调用的效率比较高，一般情况下直接采用递归定义的函数就可以满足需要，不必考虑做出非递归的函数定义

队列（queue）

- 队列（queue），或称为队，也是一种容器
  - 可存入数据元素、访问元素、删除元素
  - 这里也没有位置的概念。队列保证任何时刻可访问、删除的元素都是在此之前最早存入队列而至今未删除的那个元素
  - 队列确定了一种由存储顺序决定的访问顺序
- 队列的基本操作也是一个封闭集合，通常包括：
  - 创建空队列
  - 判断队列是否为空（还可能需要判断满）
  - 将一个元素放入队列（常称为入队，**enqueue**）
  - 从队列中删除（常称为出队，**dequeue**）一个元素
  - 取当前（最老的）元素的值（并不删除）
- 我们也可以为队列建立一个理论模型，从略

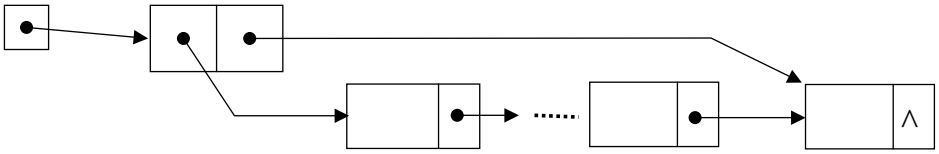
队列：特征

- 队列的特性：
  - 保证任何时刻访问或删除的元素的先进先出（**FIFO**）顺序
  - 是一种与“时间”有关的结构
  - 队列可看作（可实现为）只在一端插入另一端访问和删除的表
- 有些书籍里把队列称为先进先出（**FIFO**）表
  - 出队操作的一端称为队头
  - 入队操作的一端称为队尾



队列的链接表实现

- 用线性表的技术实现队列，就是利用元素位置的顺序关系表示入队时间的先后关系。先进先出需要在表的两端操作，实现起来比栈麻烦一些
- 首先考虑用链接表的实现，有效操作应该考虑带表尾指针的链接表
  - 这样才能保证入队/出队操作都能在 **O(1)** 时间完成
  - 如果没有表尾指针，入队就是 **O(n)** 操作，显然不理想
- 采用带表尾结点指针的链接表，后端插入为 **O(1)** 操作：



入队在表尾进行，出队在表头进行，都是 **O(1)** 时间操作。实现很简单

显然，前面相应链表的实现技术可以直接用于实现队列

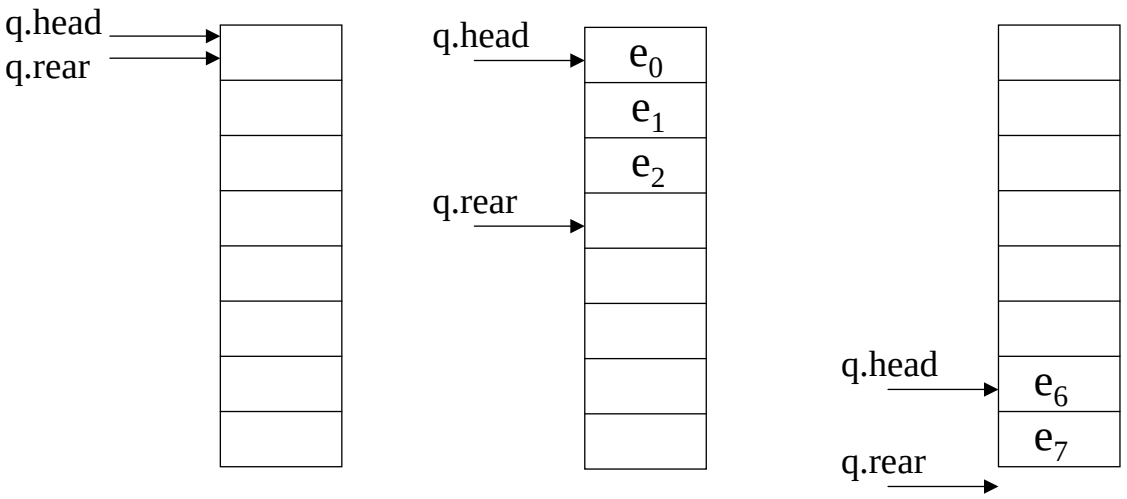
只需修改操作名，把 **append** 改为 **enqueue**，**pop** 改为 **dequeue**

队列的顺序表实现

- 现在考虑用顺序表技术实现队列
  - 假设用尾端插入实现 **enqueue**，出队操作应在表的首端进行
  - 为了维护表的完整性，每次出队操作取出首元素后，必须把它之后的元素全部前移，这样得到的是一个 **O(n)** 操作
- 反过来实现：尾端弹出元素是 **O(1)** 操作，但首端插入也是 **O(n)** 操作。这样也出现了 **O(n)** 操作，同样很不理想
- 考虑首元素出队后元素不前移，记住新队头位置。这一设计也有问题：
  - 反复入队出队，如果元素存储区固定，一定会在某次入队时出现队尾溢出表尾（表满）的情况
    - 出现这种溢出时，顺序表前部通常会有一些空闲位置
    - 这是“假性溢出”，并不是真的用完了整个元素区
  - 如果元素存储区自动增长（如 **list**），首端将留下越来越大的空区。而且这片空区永远也不会用到（完全浪费了）

队列的顺序表实现

简单实现方式有问题的示意图（设 **q** 是队列对象）：



空队列

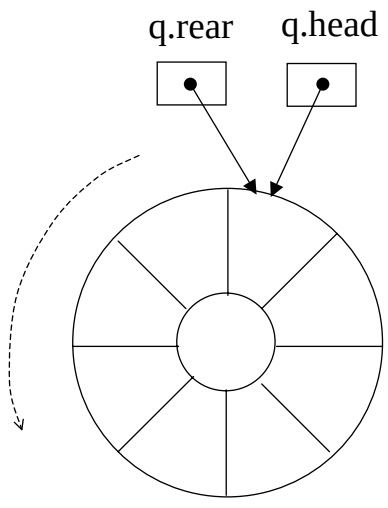
再存入就导致顺序表出界

实现方法不合适！

队列的顺序表实现

- 人们提出的一种称为“环形队列”的技术，来解决这个问题

“环形队列”（把数组看成环形）



队列空的情况

实现中的不变关系（不变式）：

q.rear 是最后元素之后空位的下标

q.head 是首元素的下标

[q.head, q.rear) 是队列中所有元素（看作按照环形排列）

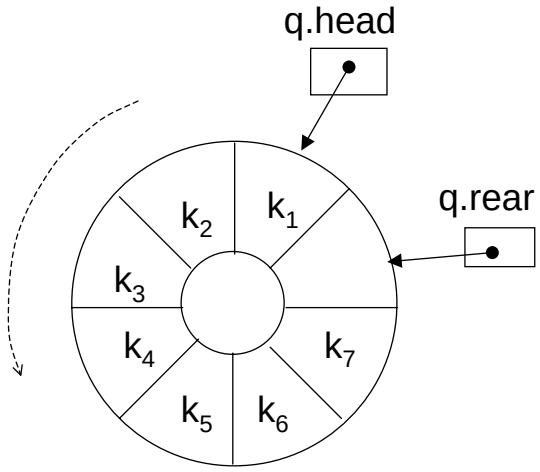
入队时，先存入，后移位

当 **q.head == q.rear** 时队列空

队列满如何判断？

条件不能与队列空判断相同

队列的顺序表实现



一种方案，队列满用下面条件判断：

(q.rear + 1) % q.len == q.head

这样做实际上空闲了一个单元

入队出队时的下标更新语句

q.head = (q.head+1) % q.len

q.rear = (q.rear + 1) % q.len

保证更新后的下标的值正确

完全可以采用其他设计，例如：

- 用 **head** 域记录队头元素位置，**elnum** 记录队中元素个数
- 队尾空位在（**q.len** 是表长）

  (q.head+q.elnum)%q.len
- 基于这两个变量实现操作，可以不空闲单元

队列的 list 实现

- 可以基于 **Python** 的 **list** 实现顺序表示的队列

  最简单的实现方法得到 **O(1)** 的 **enqueue** 和 **O(n)** 的**dequeue**
- 由于 **Python list** 不提供检查元素存储区容量的机制，我们很难利用其自动扩充元素区的机制，但可以自己做
- 首先，队列可能由于空而无法 **dequeue**，自己定义一个异常

```
class QueueUnderflow(ValueError):
    pass
```

  异常是类，自定义异常需要继承系统提供的异常。本类的体为空
- **SQueue** 类的基本考虑：
  - 用 **SQueue** 对象的一个 **list** 类型的成分 **elems** 存放队列元素
  - 用 **head** 和 **elnum** 记录首元素所在位置的下标和表中元素个数
  - 为能判断存储区满以便换一个表，需要记录表的长度，用 **len**

数据不变式

- 这里的队列实现是一个比较复杂的问题
  - 要考虑一组操作和队列对象的一组成分，其中一些操作的执行可能改变一些对象成分的取值。问题：允许怎样的改变？
  - 如果一个操作有错或与其他操作不一致，就会破坏整个对象。可见，所有操作在成分修改方面必须有统一的原则，相互合作
  - 为保证对象的完整性，各操作的实现都必须遵循这些些原则
- 为解决这类问题（一个数据结构的操作需相互协调，具有某种统一性），人们提出了“数据不变式”概念，它刻画“什么是一个完好的对象”
  - 数据不变式基于对象的成分，描述它们应满足的逻辑约束关系
  - 对象的成分取值满足数据不变式，说明这是一个状态正确的对象
- 数据不变式提出了对操作的约束和基本保证：
  - 构造对象的操作必须把对象成分设置为满足数据不变式的状态
  - 每个操作保证其对于对象成分的修改不打破数据不变式

数据不变式

- 针对下面实现，考虑的数据不变式是（用非形式的描述方式）：
  - **elems** 成分引用着队列的元素存储区，是一个 **list** 对象，**len** 成分是这个存储区的有效容量（我们并不知道该 **list** 对象的实际大小）
  - **head** 是队列首元素（当时在队列里的存入最早的那个元素）的下标，**elnum** 始终记录着队列中元素的个数
  - 队列里的元素在 **elems** 里连续存放，但需要在下标 **len** 存入元素时，操作改在下标 **0** 的位置存入
  - 在 **elnum == len** 的情况下，入队列操作将自动扩张存储区
- 下面用一个类实现一种连续表示的队列
  - **__init__** 操作建立空队列，设置对象成分保证上述不变式成立
  - 两个修改对象的变动操作都维持不变式的成立，我们将仔细检查有关情况。这样一组操作形成了一套相互协调的实现
- 前面提出过队列的其他设计，同样可以写出有关的数据不变式

队列的 list 实现

- 类定义里的几个简单方法：

```
class SQueue():
    def __init__(self, init_len = 8):
        self.len = init_len # recorded length of mem-block
        self.elems = [0] * init_len
        self.head = 0       # index of head element
        self.elnum = 0      # number of elements, initially 0
    def is_empty(self):
        return self.elnum == 0
    def first(self): # get first element, but not remove it
        if self.elnum == 0:
            raise QueueUnderflow
        return self.elems[self.head]
    # 自定义异常的使用与系统异常一样。（下页继续）
```


队列的 list 实现

- 出队列的方法很简单

```
    # 注意 head 更新的计算，
    def dequeue(self):
        if self.elnum == 0:
            raise QueueUnderflow
        e = self.elems[self.head]
        self.head = (self.head+1) % self.len
        self.elnum -= 1
        return e
```

- 入队列操作比较麻烦
  - 需要检查队列存储区是否已满
  - 存储区满时需要建一个新的，拷贝已有元素

队列的 list 实现

- 入队列方法，

```
def enqueue(self, elem):
    if self.elnum == self.len:
        self.__extend()
    self.elems[(self.head+self.elnum)%self.len] = elem
    self.elnum += 1

def __extend(self):
    old_len = self.len
    self.len *= 2
    new_elems = [0]*(self.len)
    for i in range(old_len):
        new_elems[i] = self.elems[(self.head+i)%old_len]
    self.elems, self.head = new_elems, 0
```

扩张存储区定义为一个内部过程，虽只用一次，但使程序更清晰，扩张策略也局部化。名字加两个下划线是 **Python** 内部方法惯用命名法

队列的应用

- 队列在各种计算机程序和软件里使用广泛，看几个例子
- 计算机可能连着一台打印机，可以把一些文件送去打印
  - 打印机速度有限，可能出现这样的情况：它正在打印一个文件的过程中，人们又送去一个或几个文件。对多人共享的网络打印机，很可能出现接到了很多打印任务但不能立即处理的情况
  - 打印机管理程序管理着一个缓存打印任务的队列。接到新打印任务时，如果打印机忙，该任务就被放入队列。一旦打印机完成了当前工作，管理程序就查看队列，取出最早的任务送给打印机
- 考虑网络上的一台 **Web** 服务器
  - 服务器会不断接到来自网络的网页请求，这时它应该设法找到或做出所需要的页面，发送给提出请求的网络客户
  - 来自网络的请求在速率上会有很大波动，如果瞬时请求很多，服务器就会把来不及处理的请求放入一个队列。一个处理器（或处理线程）完成当时的工作后，就会到队列里取走一个未处理请求

队列的应用

- Windows 系统是围绕着“消息”概念和机制构造和活动的（称为是消息驱动的系统），所有应用程序都需要围绕消息构造起来的
  - 各种活动（窗口界面操作，各种输入输出，程序活动）都可能/可以产生各种消息，要求某些系统程序或用户程序响应
  - Windows 系统维护着一些消息队列，保存接到的各种消息；消息分发机制检查消息里的信息，把它们分发给相应程序；程序在处理消息的过程中又可能生成新的消息
  - 在 Windows 系统里工作的每个程序都有一个隐含的消息队列。程序里有一个消息处理循环，每次循环检查自己的消息队列，如果没有消息就进入等待，有消息就取出来处理
- 计算机里不同处理器或处理进程（线程）之间的通讯也可能需要消息队列作为缓冲（这种方式称为异步通讯）
  - 通讯服务软件把发给一个进程的消息放入该进程的消息队列
  - 进程需要消息时查看自己的消息队列，根据情况取出处理或等待

队列应用：离散事件模拟

- 队列的另一种常见应用是做离散事件系统模拟
- 离散事件系统的典型情况
  - 被模拟的（真实世界的）系统的行为表现为一些活动，各种活动进行中又可能产生新的活动。这些活动都需要处理，最简单最常见的处理方式是采用“先发生先处理”的工作原则
  - 由于事件的产生和处理之间存在着速度上的波动和差异，因此系统里可能存在一些已经在等待处理的活动。模拟这种系统，就需要用队列记录正在等待的序列
- 离散事件系统的实例
  - 银行等待服务的顾客、服务席位和服务时间
  - 高速公路收费站通道和服务安排
  - 大楼电梯系统设计和安排
  - 计算机网络中的各种服务系统

队列应用：离散事件模拟

- 做系统模拟，是希望通过计算机程序的运行模拟真实系统的活动，以理解真实系统运行时的行为，或在未实现系统之前做出一些设计决策
- 在实现这种模拟系统时，一般而言，需要
  - 通过调查或设计，选择一批模拟参数（例如，顾客抵达的频度）
  - 引入一些随机因素，反映真实世界中的各种非确定性情况（例如，确定顾客到达约为每2分钟一人，正负1分钟，有随机性）
  - 用一个或一批队列保存各种待处理活动（例如正等待的顾客）
- 在生成的活动中保存一些反映真实世界情况的信息（例如，顾客到达的时间，接受服务的开始时间，离开的时间等），以便所实现的模拟系统最后能完成一些统计工作，得到有参考价值的模拟结果（例如：平均等待时间等），用以指导系统设计
- 离散系统模拟还常要考虑另外的一些顺序因素，如服务完成的时间等。我们将在后面讨论“优先队列”之后再考虑离散事件模拟系统的实例